Hardware Memory Models: x86-TSO

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Agenda

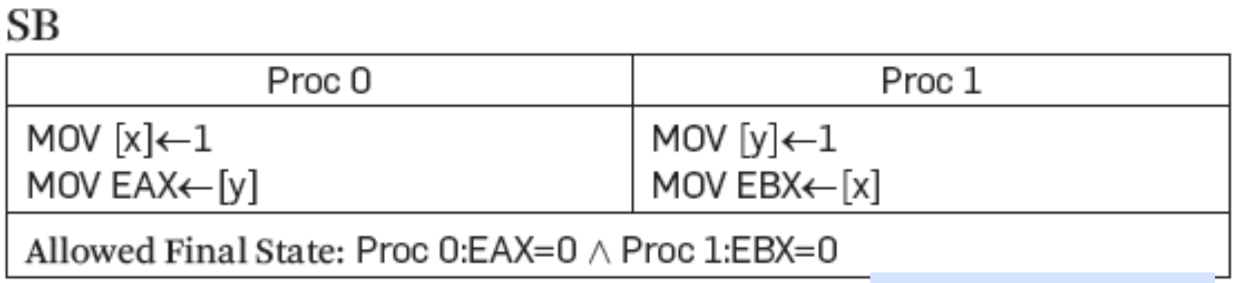
- Last class
  - —overview of memory models
- Today: hardware implementation of TSO memory model
  - —Sewall et al. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. CACM 53(7):89-97. http://doi.acm.org/10.1145/1785414.1785443

Requirements for Multithreaded HW

- Reliable, high-performance parallel code
  - —operating system kernel
  - —libraries
    - – language runtime systems
    - – synchronization primitives
    - – concurrent data structures
- Compilers for concurrent languages

Challenges - I

- Multiprocessors typically do not provide sequentially consistent memory
  - —why? optimizations for performance!
    - e.g., store buffers to hide write latency, speculative execution
- Multithreaded codes often observe a relaxed memory model
  - —threads have only loosely consistent views of shared memory
    - e.g., visible evidence of store buffering on x86



SB

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

Figure credit: [1]

- Understanding relaxed models is necessary for writing correct parallel software

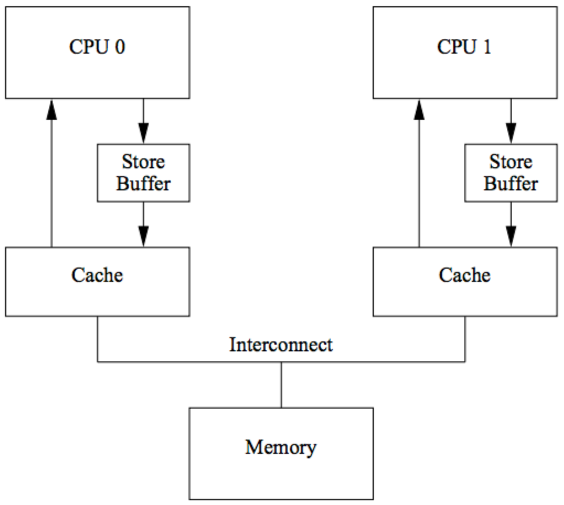


Figure 5: Caches With Store Buffers

Figure credit: [2]

Store Buffers and Store Forwarding

Problem: without store forwarding a memory location may appear to have multiple values

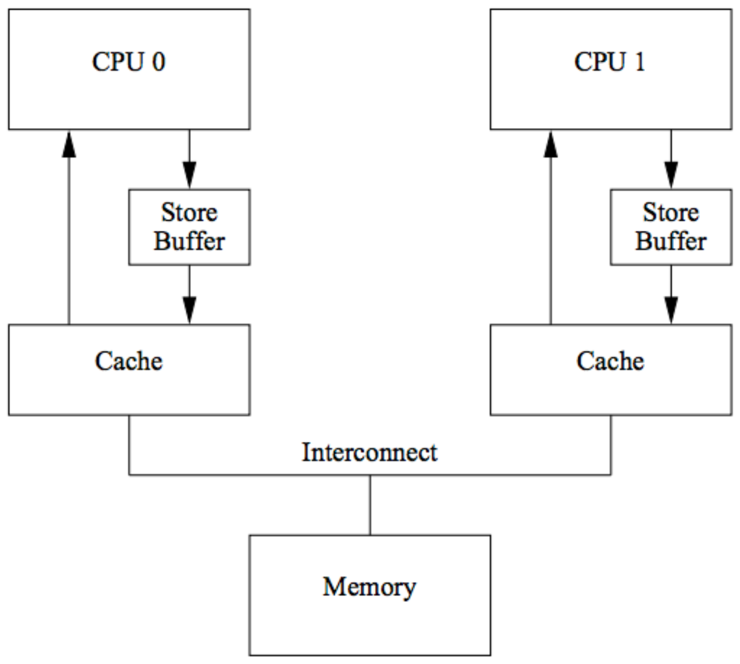


Figure 5: Caches With Store Buffers

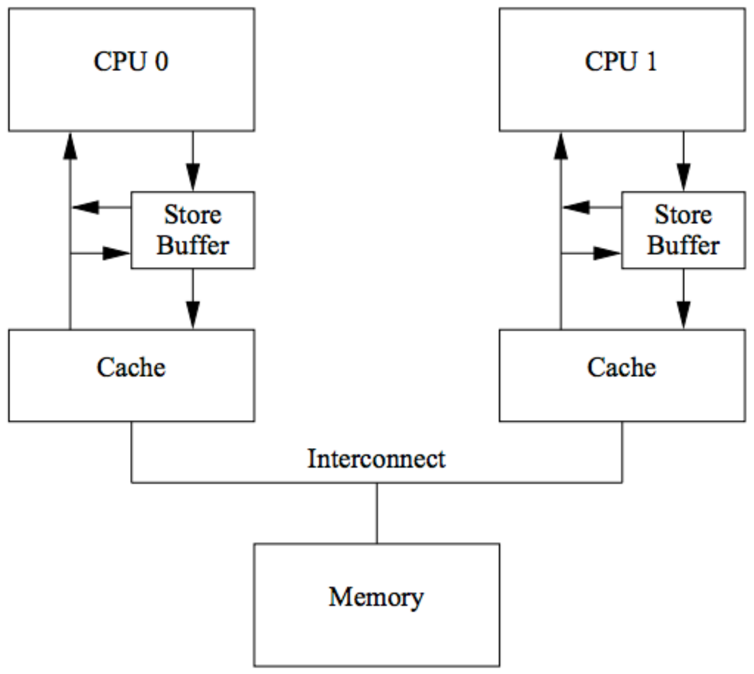


Figure 6: Caches With Store Forwarding

Figure credit: [2]

Store Buffers

Problem: without store forwarding a memory location may appear to have multiple values

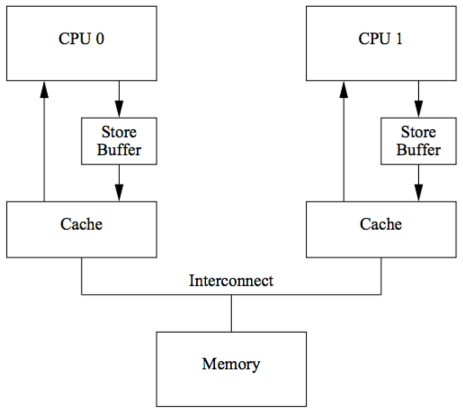


Figure 5: Caches With Store Buffers

```
a = 1;
b = a + 1;
assert(b == 2);
```

One would not expect the assertion to fail. However, if one were foolish enough to use the very simple architecture shown in Figure 5, one would be surprised. Such a system could potentially see the following sequence of events:

1. CPU 0 starts executing the `a=1`.
2. CPU 0 looks "a" up in the cache, and finds that it is missing.
3. CPU 0 therefore sends a "read invalidate" message in order to get exclusive ownership of the cache line containing "a".
4. CPU 0 records the store to "a" in its store buffer.
5. CPU 1 receives the "read invalidate" message, and responds by transmitting the cache line and removing that cacheline from its cache.
6. CPU 0 starts executing the `b=a+1`.
7. CPU 0 receives the cache line from CPU 1, which still has a value of zero for "a".
8. CPU 0 loads "a" from its cache, finding the value zero.
9. CPU 0 applies the entry from its store queue to the newly arrived cache line, setting the value of "a" in its cache to one.
10. CPU 0 adds one to the value zero loaded for "a" above, and stores it into the cache line containing "b" (which we will assume is already owned by CPU 0).
11. CPU 0 executes `assert(b==2)`, which fails.

Figure credit: [2]

Store Buffers with Forwarding

Problem: without store forwarding a memory location may appear to have multiple values

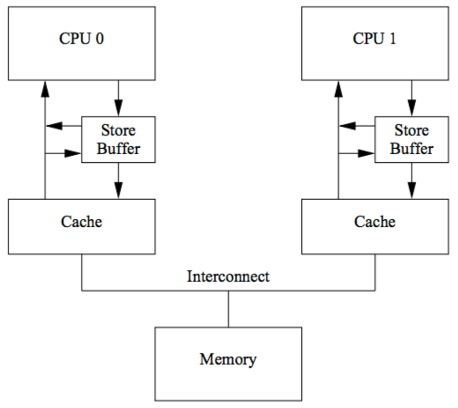


Figure 6: Caches With Store Forwarding

```
void foo(void)
{
  a = 1;
  b = 1;
}

void bar(void)
{
  while (b == 0) continue;
  assert(a == 1);
}
```

Suppose CPU 0 executes foo() and CPU 1 executes bar(). Suppose further that the cache line containing "a" resides only in CPU 1's cache, and that the cache line containing "b" is owned by CPU 0. Then the sequence of operations might be as follows:

1. CPU 0 executes `a=1`. The cache line is not in CPU 0's cache, so CPU 0 places the new value of "a" in its store buffer and transmits a "read invalidate" message.
2. CPU 1 executes `while(b==0)continue`, but the cache line containing "b" is not in its cache. It therefore transmits a "read" message.
3. CPU 0 executes `b=1`. It already owns this cache line (in other words, the cache line is already in either the "modified" or the "exclusive" state), so it stores the new value of "b" in its cache line.
4. CPU 0 receives the "read" message, and transmits the cache line containing the now-updated value of "b" to CPU 1, also marking the line as "shared" in its own cache.
5. CPU 1 receives the cache line containing "b" and installs it in its cache.
6. CPU 1 can now finish executing `while(b==0) continue`, and since it finds that the value of "b" is 1, it proceeds to the next statement.
7. CPU 1 executes the `assert(a==1)`, and, since CPU 1 is working with the old value of "a", this assertion fails.
8. CPU 1 receives the "read invalidate" message, and transmits the cache line containing "a" to CPU 0 and invalidates this cache line from its own cache. But it is too late.
9. CPU 0 receives the cache line containing "a" and applies the buffered store just in time to fall victim to CPU 1's failed assertion.

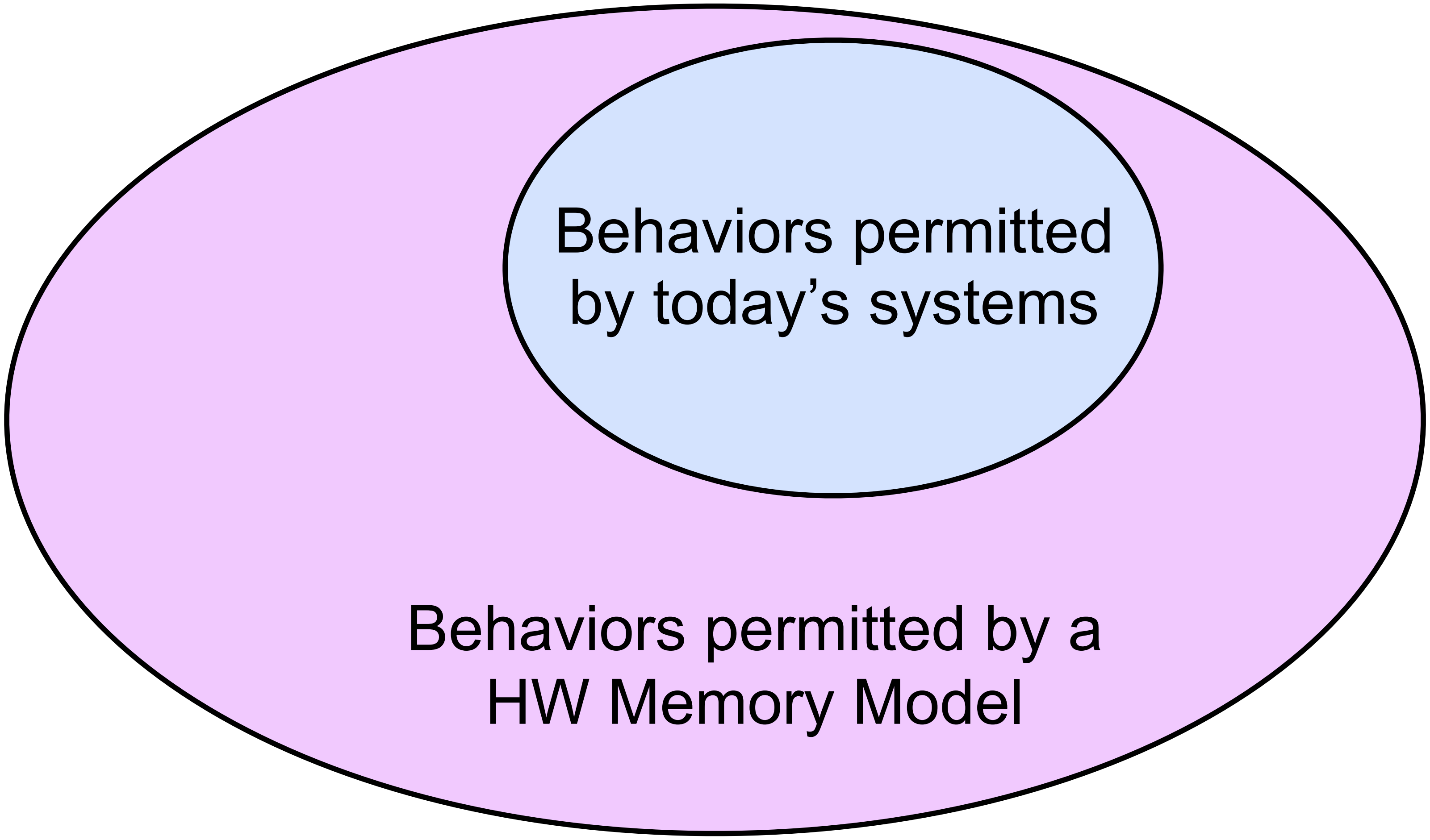
Figure credit: [2]

Challenges - II

- Different processor families use different relaxed models
- Commodity vendors often specify memory models in ambiguous, informal prose
  - —poor medium for loose specifications
    - – inevitably ambiguous
    - – sometimes wrong
  - —example: Intel SDM rev. 22 Nov 2006: “processor ordering”
    - – no examples
  - —cause for confusion: spin lock optimization? (LKM 1999)
- Major and subtle differences between processor families
  - —what non-SC behaviors they permit
  - —memory barriers and synchronization instructions they provide

Architectural Specifications

- **Specify what programmers can rely upon**
- **Architectural specifications are "loose" to cover past and future implementations**


Behaviors permitted by today's systems
Behaviors permitted by a HW Memory Model

Memory Models for x86 Processors

- **Problem: some prior Intel and AMD specifications**
  - **—contain serious ambiguities**
  - **—are arguably too weak for writing programs**
  - **—are simply unsound with respect to actual hardware**
  - **—provide no basis for formally reasoning about programs**

example next slide

- **Contribution: new x86-TSO programmer's model**
  - **—TSO = total store order**
  - **—suffers from none of the aforementioned problems**
  - **—provides intuitive abstract machine, accessible to programmers**
  - **—is mathematically precise: rigorously defined in HOL4**
    - **memory model + semantics for machine instructions enables formal reasoning about program behavior**
- **How is this useful?**
  - **—guides intuition of systems programmers developing software for multithreaded systems**

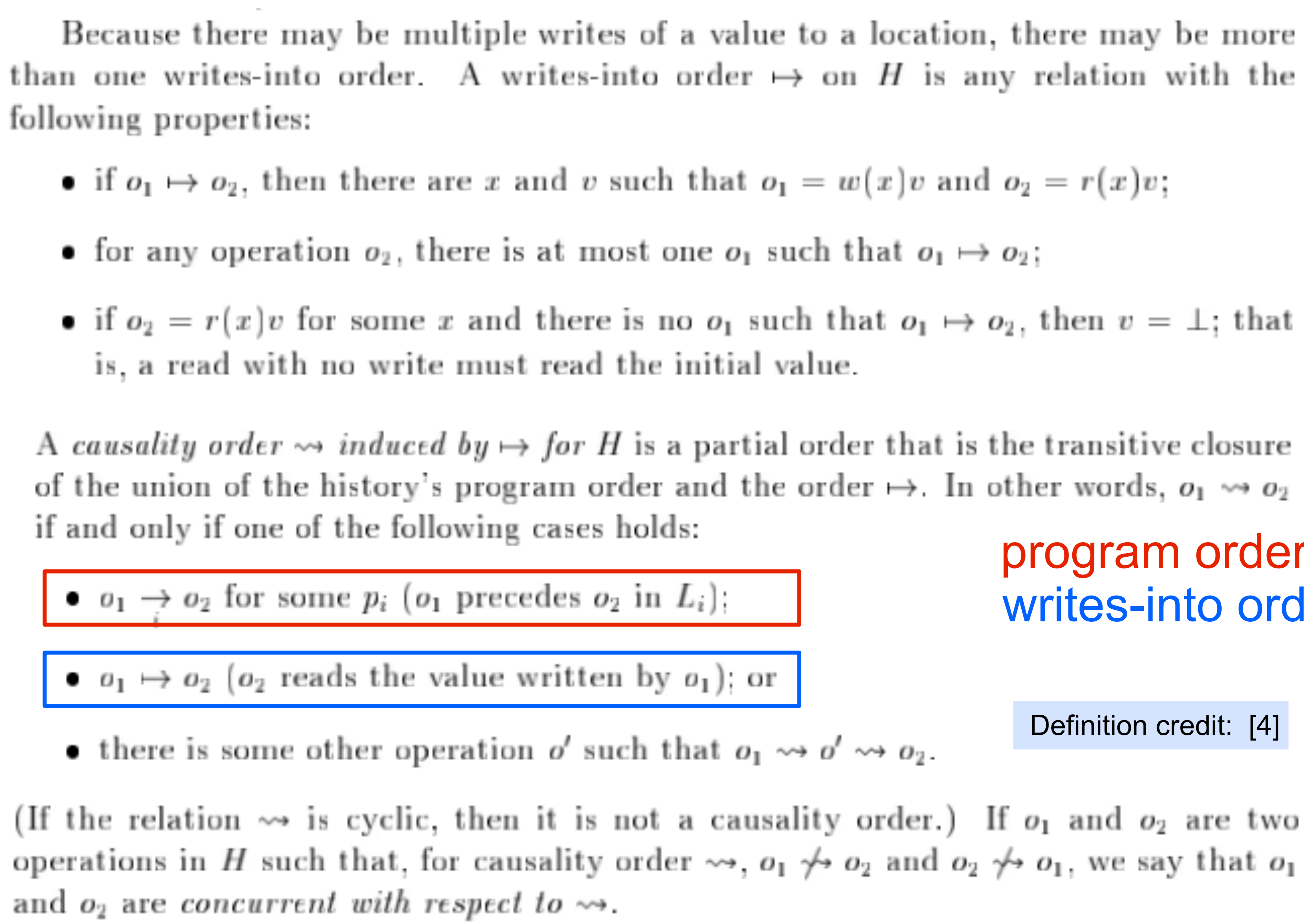
x86 Fences

- **Definitions**
  - —LFENCE: load fence
  - —SFENCE: store fence
  - —MFENCE: memory fence (strongest x86 memory barrier)
- **Operation**
  - —reads cannot pass LFENCE and MFENCE instructions
  - —writes cannot pass SFENCE and MFENCE instructions

Causal Consistency

- Processes in a system agree on the relative ordering of operations that are causally related
  - —memory ordering obeys causality
  - —respects transitive visibility
- Defined by
  - —program order
  - —writes into order
- Use of causal consistency
  - —Intel White Paper, August 2007
    - – allows causal consistency implicitly
  - —AMD Architecture Programmer’s Manual 3.14, Sept 2007
    - – allows causal consistency explicitly

Causal Consistency



Because there may be multiple writes of a value to a location, there may be more than one writes-into order. A writes-into order $\mapsto$ on H is any relation with the following properties:

- if $o_1 \mapsto o_2$, then there are x and v such that $o_1 = w(x)v$ and $o_2 = r(x)v$;
- for any operation o_2, there is at most one o_1 such that $o_1 \mapsto o_2$;
- if $o_2 = r(x)v$ for some x and there is no o_1 such that $o_1 \mapsto o_2$, then $v = \perp$; that is, a read with no write must read the initial value.

A *causality order* $\rightsquigarrow$ *induced by* $\mapsto$ *for* H is a partial order that is the transitive closure of the union of the history's program order and the order $\mapsto$. In other words, $o_1 \rightsquigarrow o_2$ if and only if one of the following cases holds:

- $o_1 \underset{i}{\rightarrow} o_2$ for some p_i (o_1 precedes o_2 in L_i);
- $o_1 \mapsto o_2$ (o_2 reads the value written by o_1); or
- there is some other operation o' such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$.

(If the relation $\rightsquigarrow$ is cyclic, then it is not a causality order.) If o_1 and o_2 are two operations in H such that, for causality order $\rightsquigarrow$, $o_1 \not\rightsquigarrow o_2$ and $o_2 \not\rightsquigarrow o_1$, we say that o_1 and o_2 are *concurrent with respect to* $\rightsquigarrow$.

program order

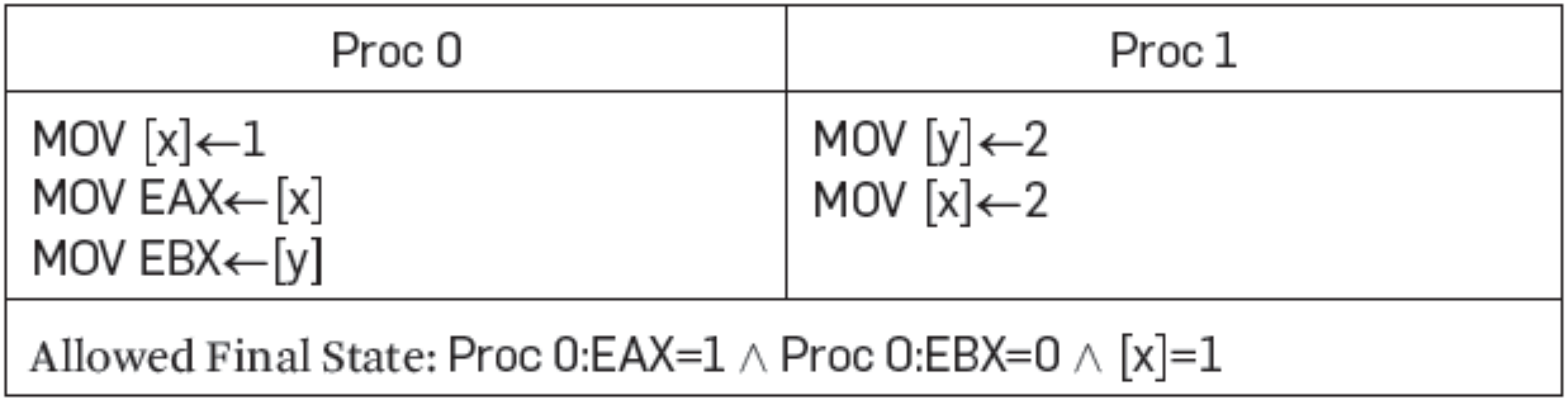
writes-into order

Definition credit: [4]

Causal Consistency (IWP/AMD3.14/x86-CC)

CC:
mov x 1 must precede the mov eax that reads it (writes into order)
mov y 2 must precede mov x 2 (program order)

n6



Proc 0	Proc 1
MOV [x]←1 MOV EAX←[x] MOV EBX←[y]	MOV [y]←2 MOV [x]←2
Allowed Final State: Proc 0:EAX=1 ∧ Proc 0:EBX=0 ∧ [x]=1	

—how?

cc : Forbid; tso : Allow

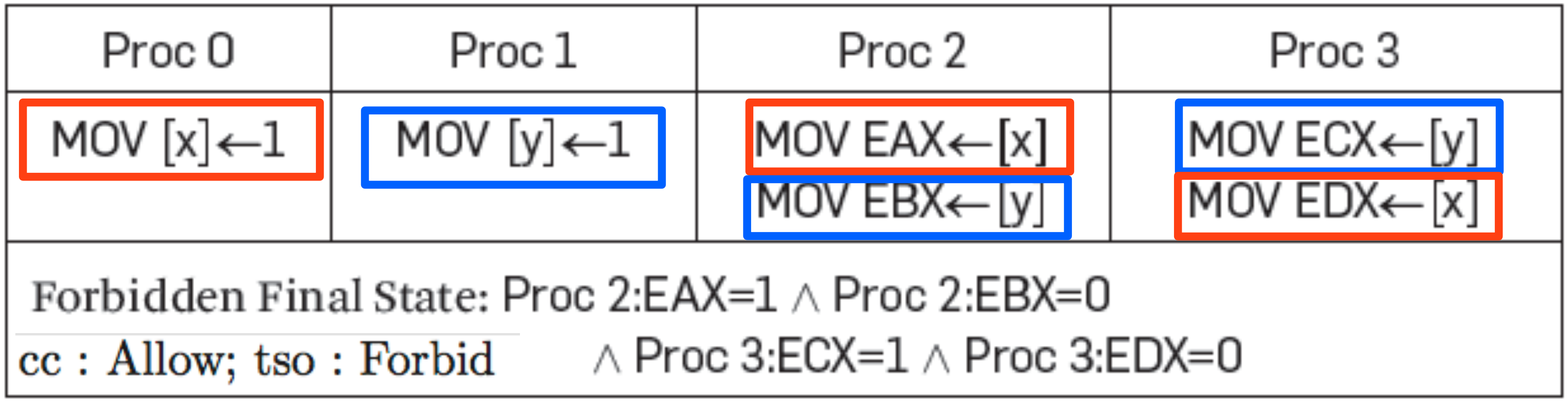
- P1 write of [y]=2 is buffered
- P0 buffers its write of [x]=1, reads [x]=1 from its store buffer, and reads [y]=0 from main memory
- P1 buffers its [x]=2 write, flushes its buffered [y]=2, [x]=2 writes to memory
- P0 flushes its [x]=1 write to memory.

Figure credit: [1]

Causal Consistency (IWP/AMD3.14/x86-CC)

- **Problem: causal consistency is too weak for programmers**
  - —admits the following inconsistent view of independent writes

IRIW

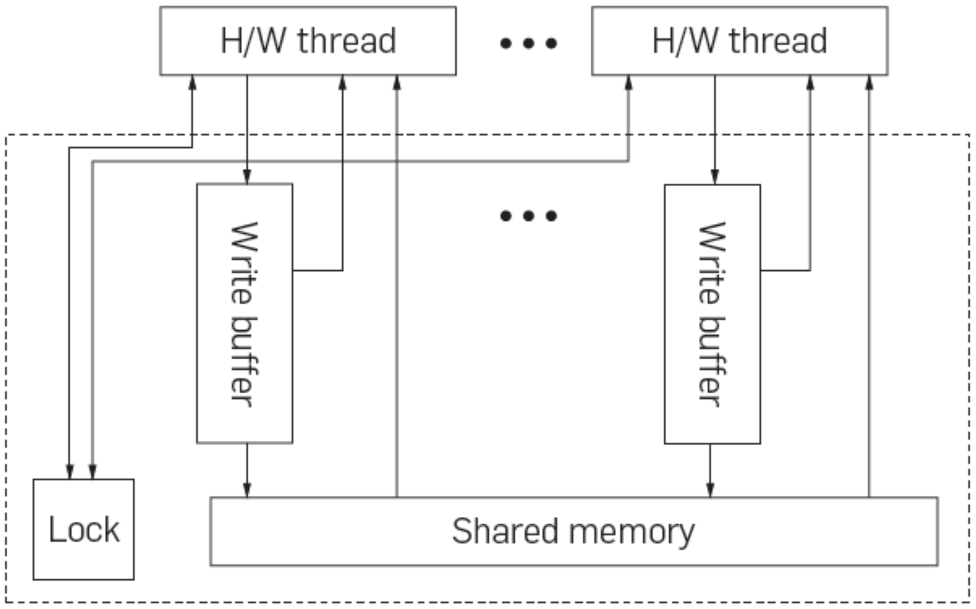


Proc 0	Proc 1	Proc 2	Proc 3
MOV [x]←1	MOV [y]←1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Forbidden Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0 cc : Allow; tso : Forbid			

  - —ordering inconsistencies can arise if store buffers are shared between some but not all threads
  - —would need to use LOCK instead of MFENCE to recover SC
  - —appears looser than behavior of implemented processors

Figure credit: [1]

x86-TSO Programmer's Model



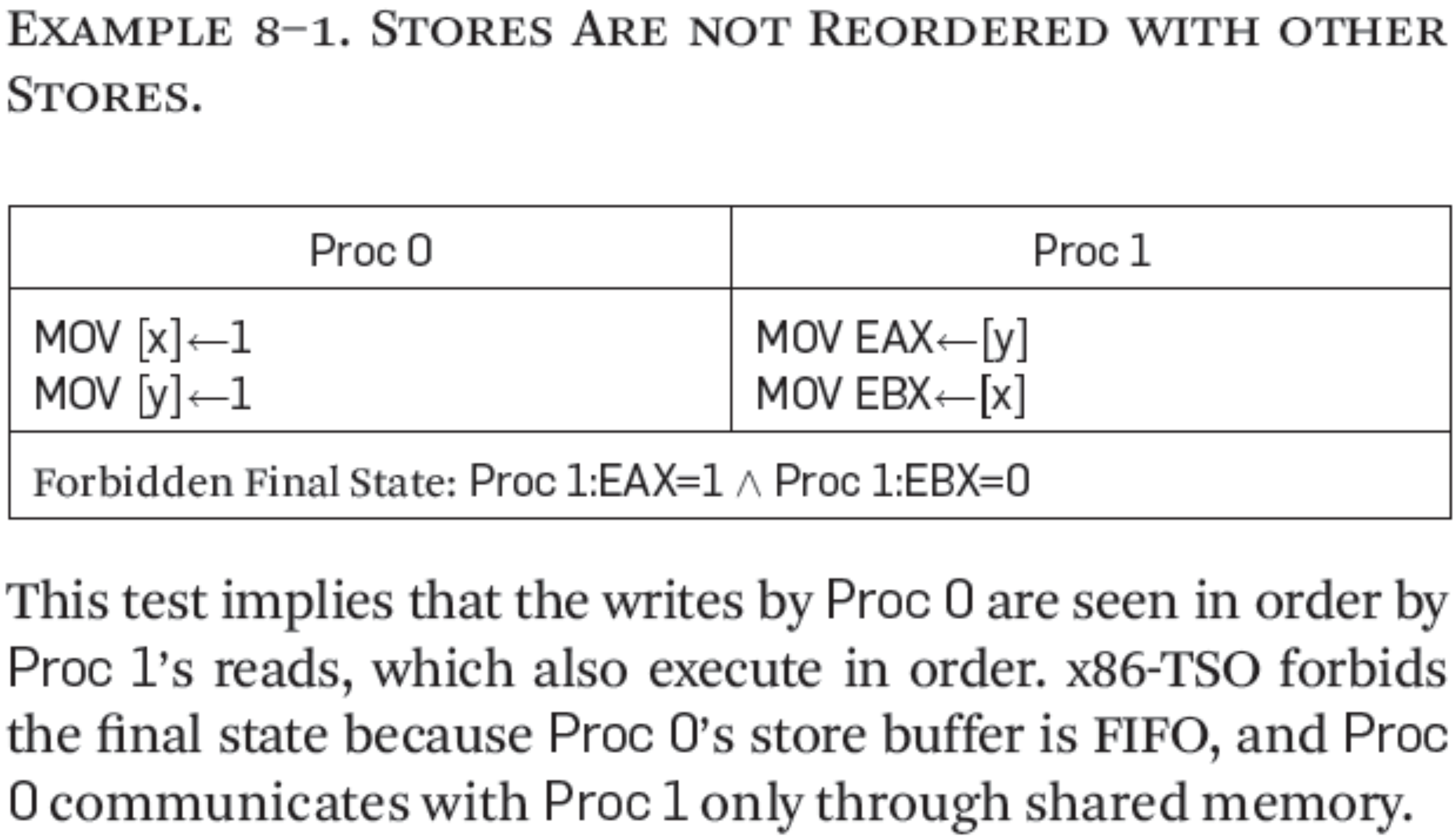
- **Store buffers are FIFO, a reading thread must read its most recent buffered write, or if none present, value from memory**
- **MFENCE flushes a thread's store buffer**
- **LOCK'd instruction (LOCK is a modifier that applies to other instructions)**
  - **—thread must obtain global lock**
  - **—after instruction, thread flushes its store buffer**
  - **—no other thread can read while global lock is held**
- **A buffered write can propagate to memory at any time, except when another thread holds the global lock**

Figure credit: [1]

Scope of x86-TSO

- Programs using cacheable, write-back memory
- Without
  - —exceptions
  - —misaligned accesses
  - —non-temporal operations (which avoid updating L1 cache)
  - —self-modifying code
  - —page table changes

x86-TSO Behaviors - I



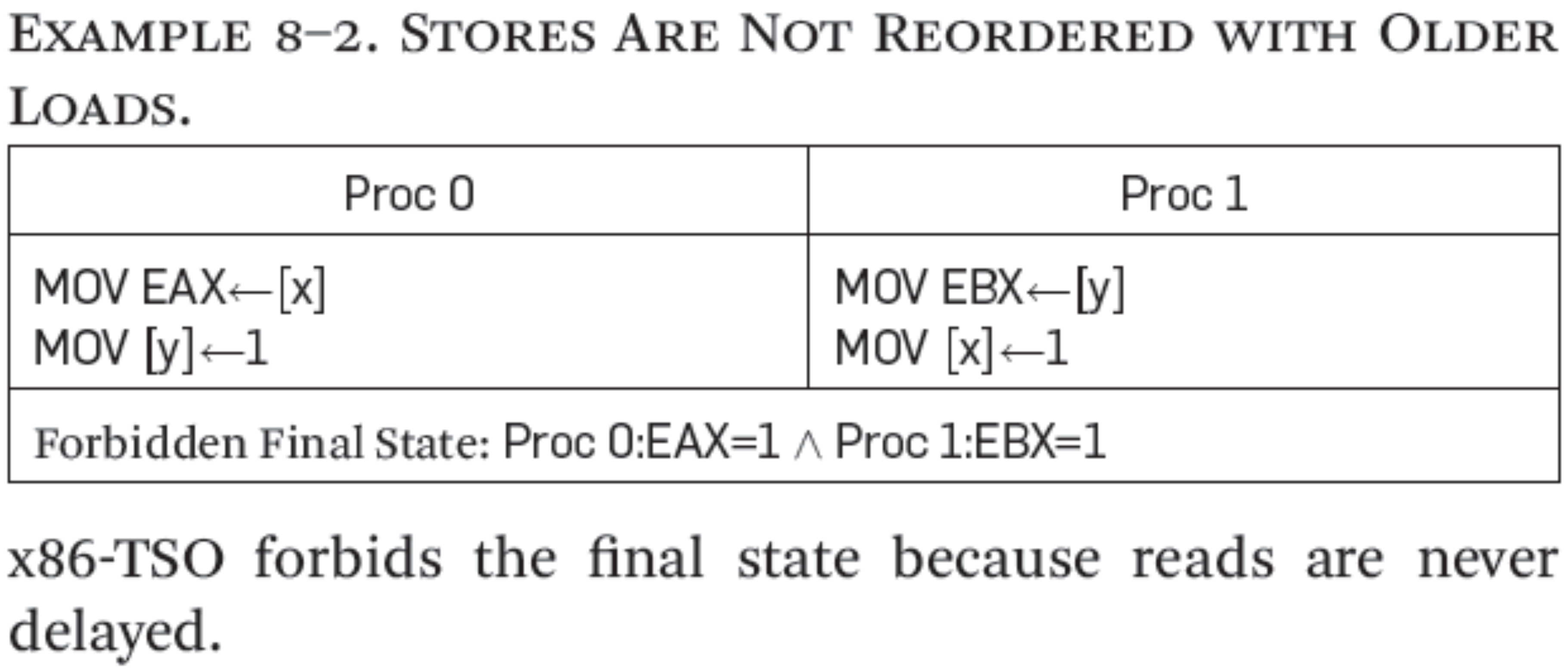
EXAMPLE 8–1. STORES ARE NOT REORDERED WITH OTHER STORES.

Proc 0	Proc 1
MOV [x]←1 MOV [y]←1	MOV EAX←[y] MOV EBX←[x]
Forbidden Final State: Proc 1:EAX=1 ∧ Proc 1:EBX=0	

This test implies that the writes by Proc 0 are seen in order by Proc 1's reads, which also execute in order. x86-TSO forbids the final state because Proc 0's store buffer is FIFO, and Proc 0 communicates with Proc 1 only through shared memory.

Figure credit: [1]

x86-TSO Behaviors - II



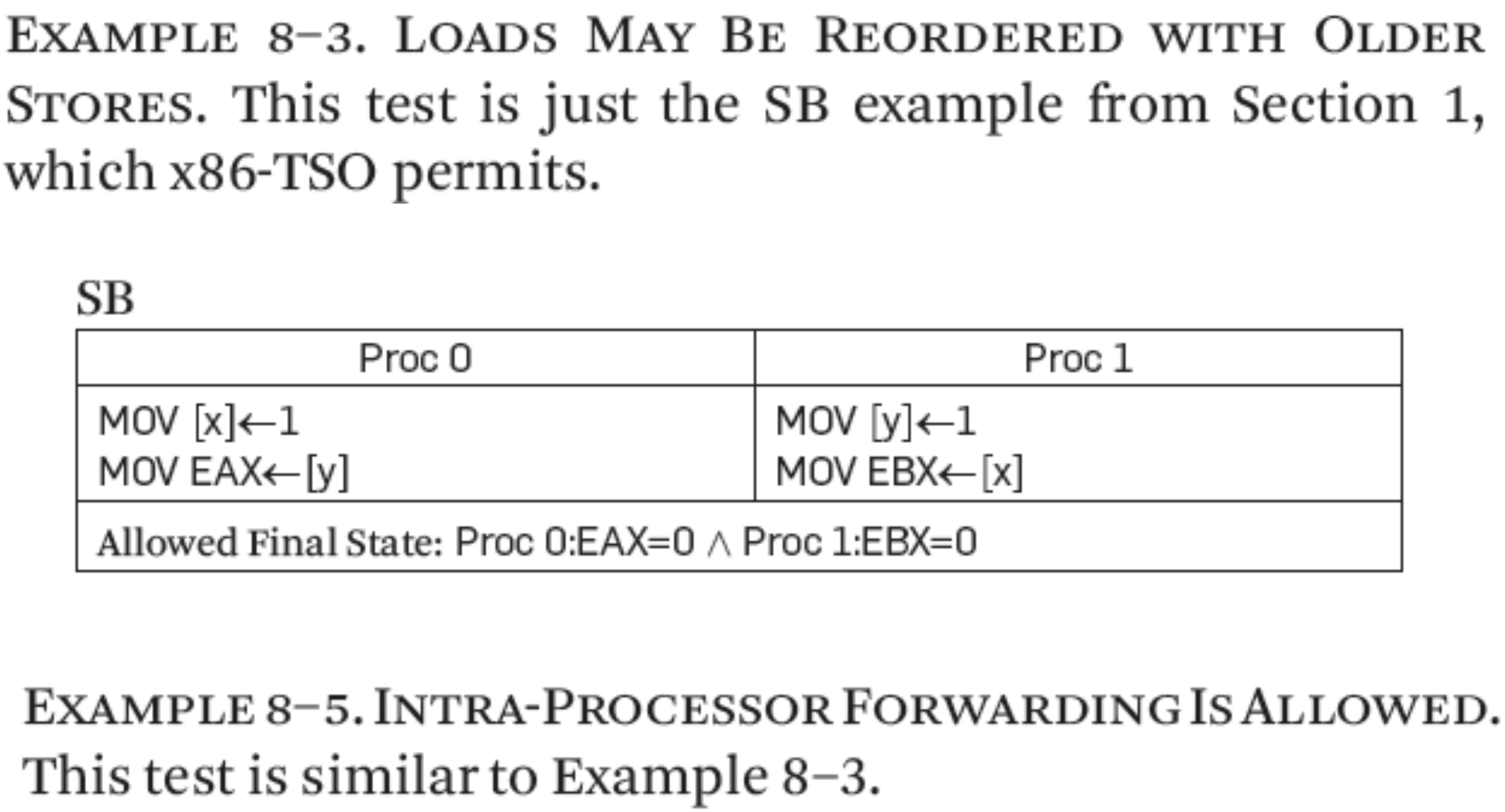
EXAMPLE 8–2. STORES ARE NOT REORDERED WITH OLDER LOADS.

Proc 0	Proc 1
MOV EAX←[x] MOV [y]←1	MOV EBX←[y] MOV [x]←1
Forbidden Final State: Proc 0:EAX=1 ∧ Proc 1:EBX=1	

x86-TSO forbids the final state because reads are never delayed.

Figure credit: [1]

x86-TSO Behaviors - III



EXAMPLE 8–3. LOADS MAY BE REORDERED WITH OLDER STORES. This test is just the SB example from Section 1, which x86-TSO permits.

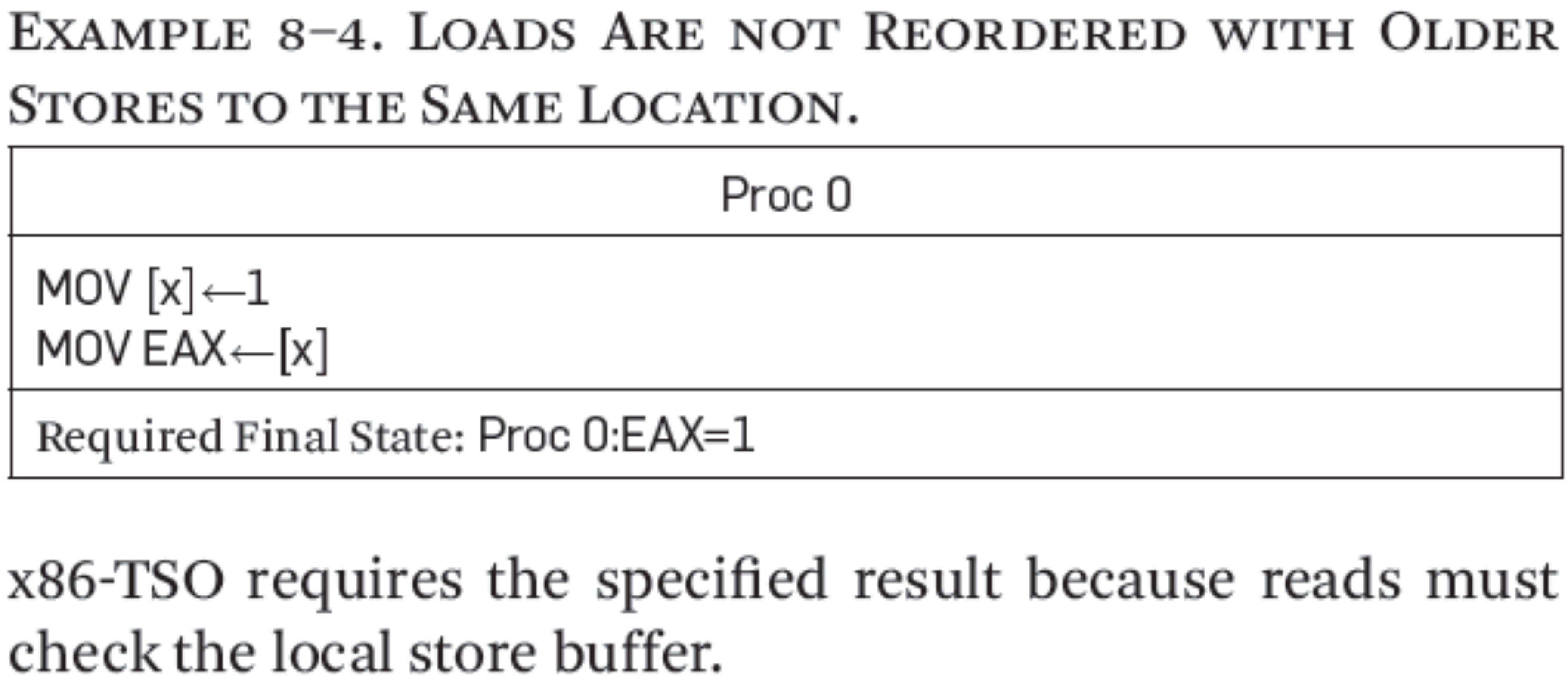
SB

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

EXAMPLE 8–5. INTRA-PROCESSOR FORWARDING IS ALLOWED. This test is similar to Example 8–3.

Figure credit: [1]

x86-TSO Behaviors - IV



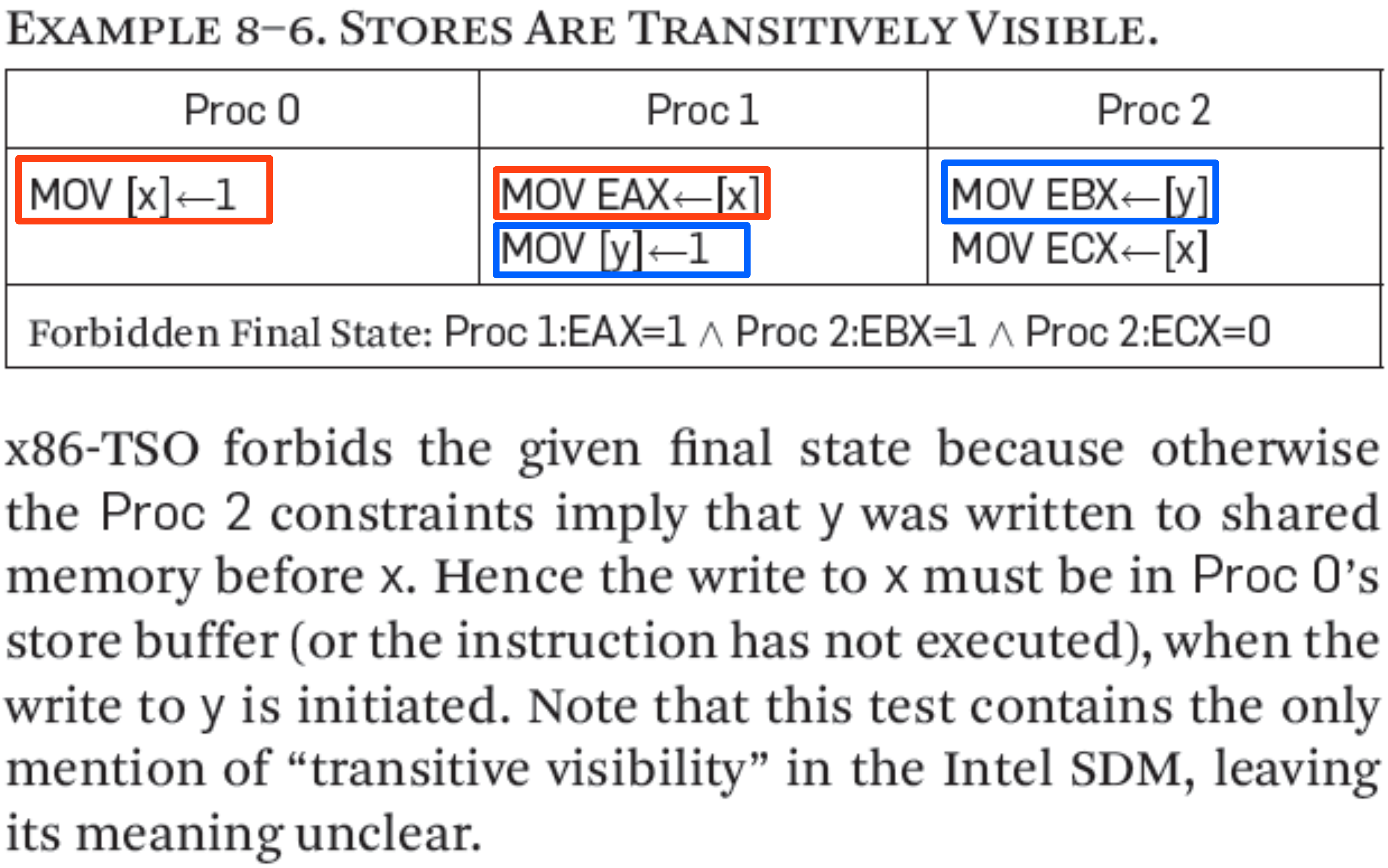
EXAMPLE 8–4. LOADS ARE NOT REORDERED WITH OLDER STORES TO THE SAME LOCATION.

Proc 0
MOV [x]←1 MOV EAX←[x]
Required Final State: Proc 0:EAX=1

x86-TSO requires the specified result because reads must check the local store buffer.

Figure credit: [1]

x86-TSO Behaviors - V



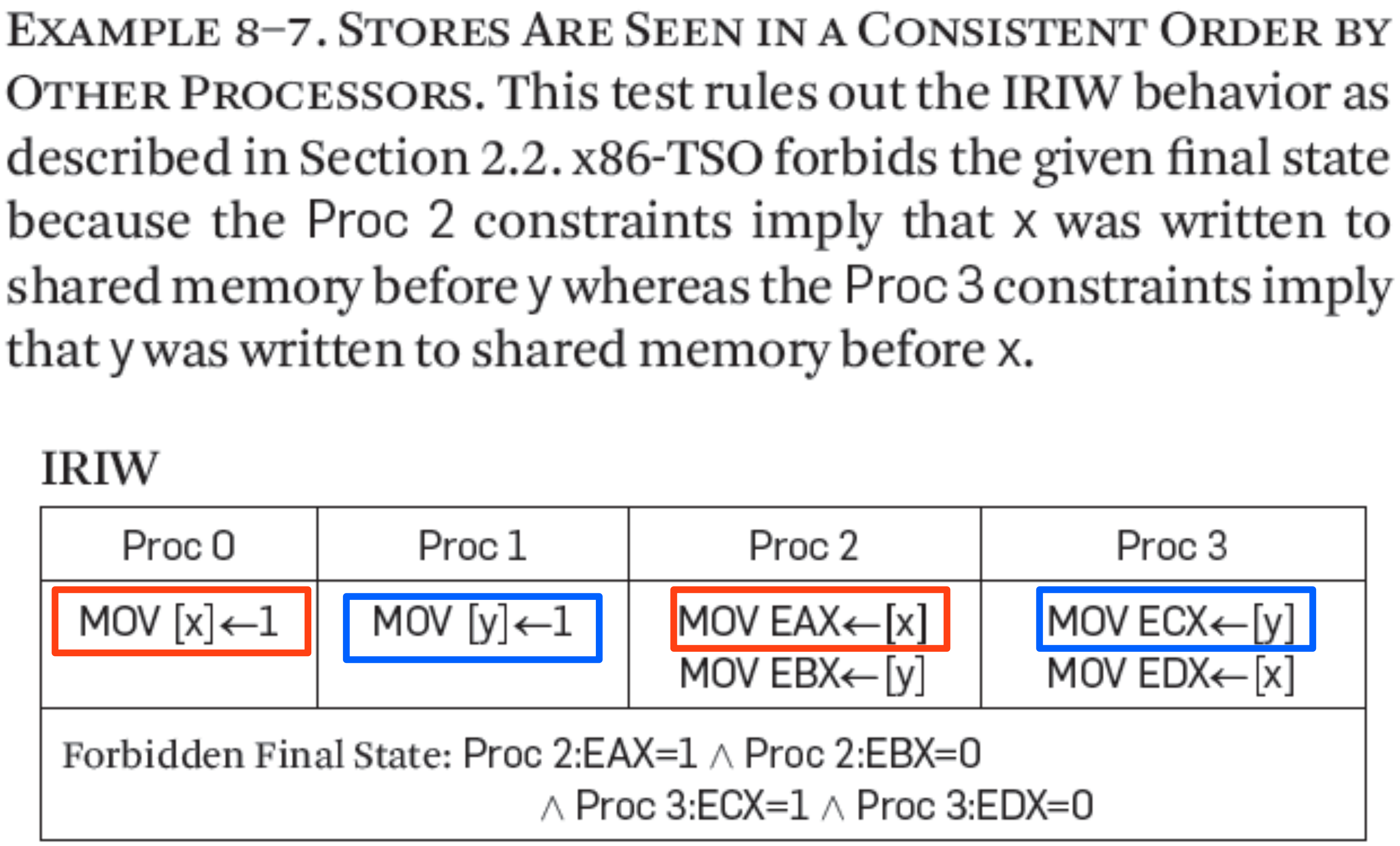
EXAMPLE 8–6. STORES ARE TRANSITIVELY VISIBLE.

Proc 0	Proc 1	Proc 2
MOV [x]←1	MOV EAX←[x] MOV [y]←1	MOV EBX←[y] MOV ECX←[x]
Forbidden Final State: Proc 1:EAX=1 ∧ Proc 2:EBX=1 ∧ Proc 2:ECX=0		

x86-TSO forbids the given final state because otherwise the Proc 2 constraints imply that y was written to shared memory before x. Hence the write to x must be in Proc 0's store buffer (or the instruction has not executed), when the write to y is initiated. Note that this test contains the only mention of "transitive visibility" in the Intel SDM, leaving its meaning unclear.

Figure credit: [1]

x86-TSO Behaviors - VI



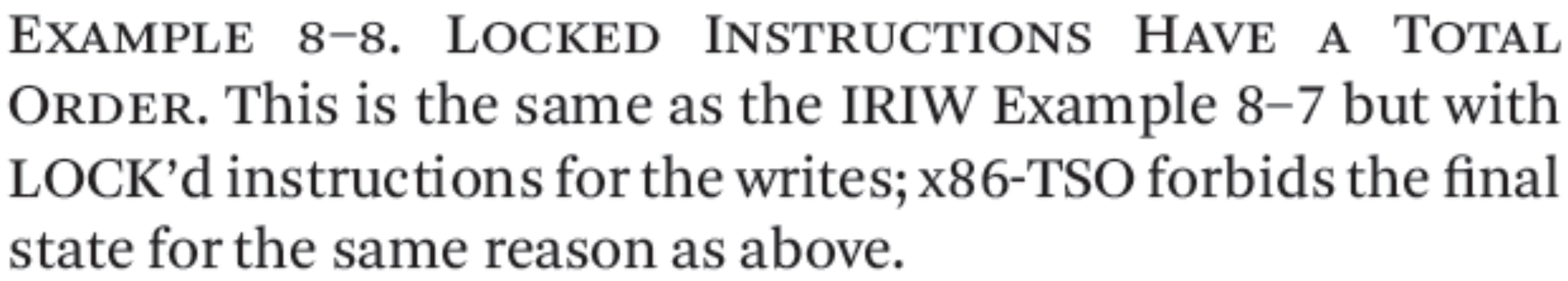
EXAMPLE 8–7. STORES ARE SEEN IN A CONSISTENT ORDER BY OTHER PROCESSORS. This test rules out the IRIW behavior as described in Section 2.2. x86-TSO forbids the given final state because the Proc 2 constraints imply that x was written to shared memory before y whereas the Proc 3 constraints imply that y was written to shared memory before x.

IRIW

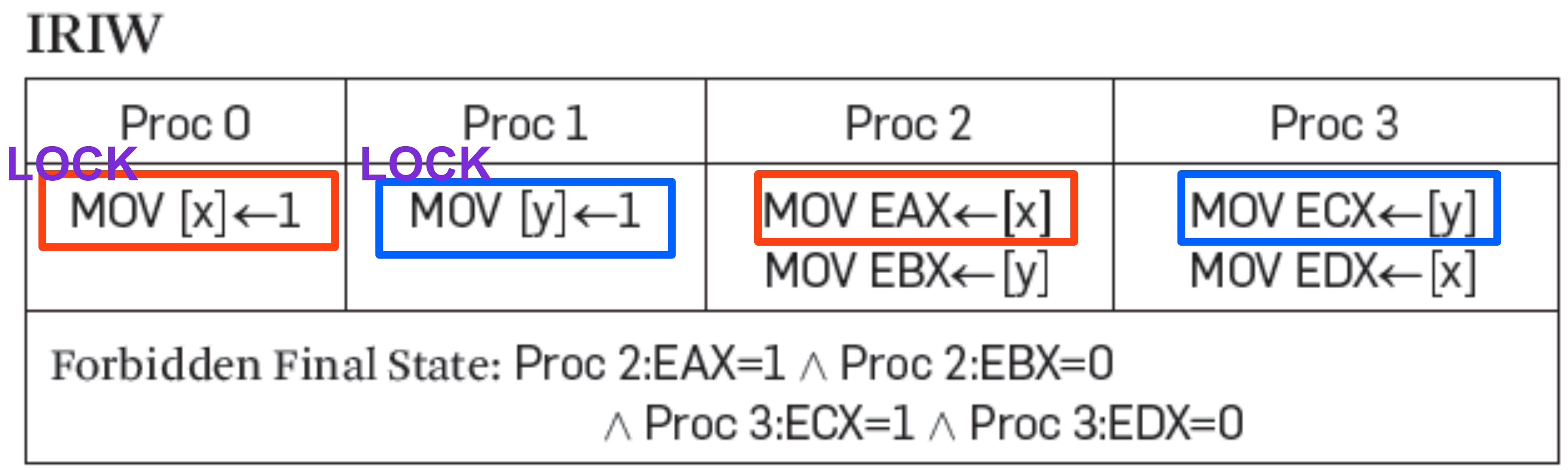
Proc 0	Proc 1	Proc 2	Proc 3
MOV [x]←1	MOV [y]←1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Forbidden Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0			

Figure credit: [1]

x86-TSO Behaviors - VII



EXAMPLE 8–8. LOCKED INSTRUCTIONS HAVE A TOTAL ORDER. This is the same as the IRIW Example 8–7 but with LOCK'd instructions for the writes; x86-TSO forbids the final state for the same reason as above.

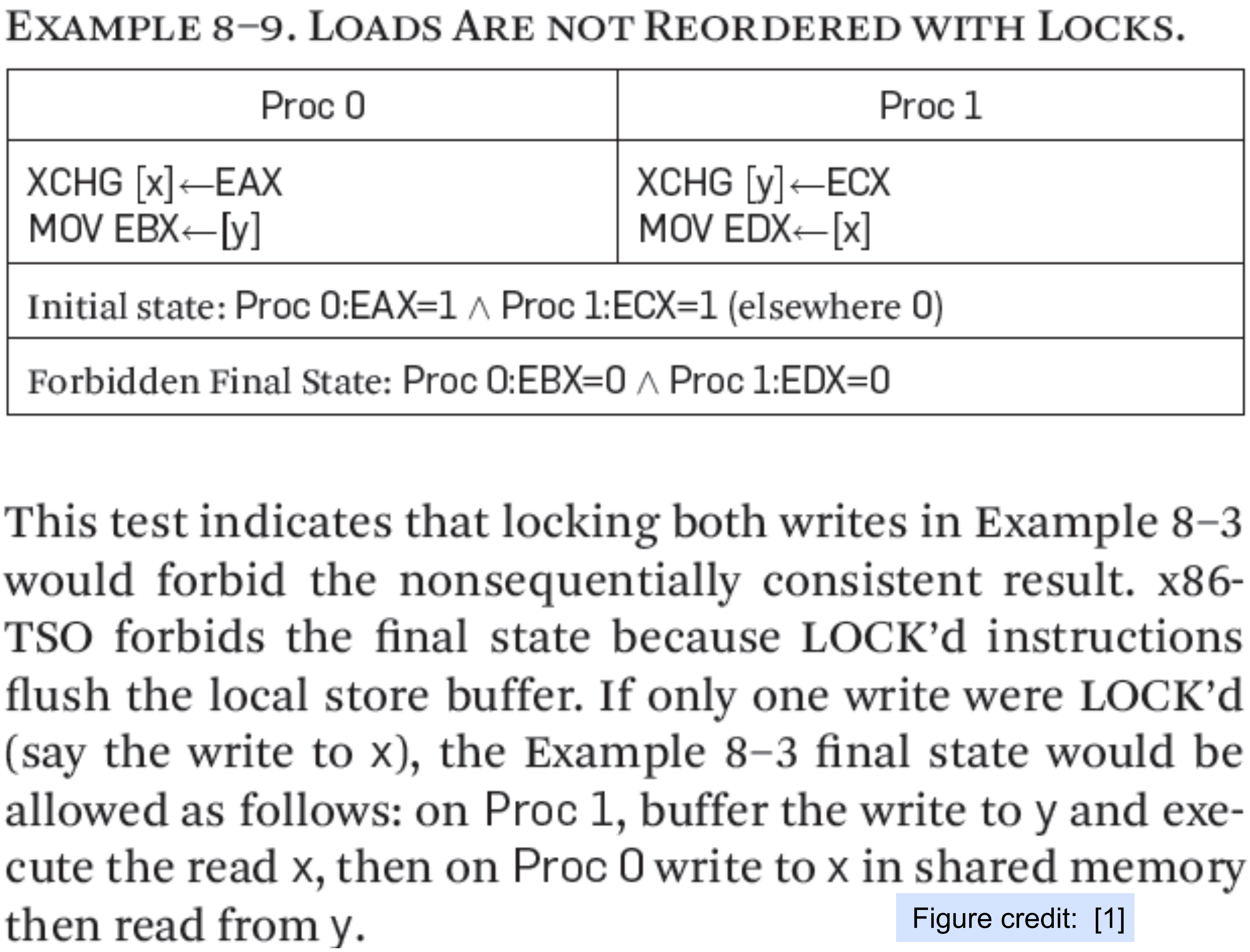


IRIW

Proc 0	Proc 1	Proc 2	Proc 3
LOCK MOV [x]←1	LOCK MOV [y]←1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Forbidden Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0			

Figure credit: [1]

x86-TSO Behaviors - VIII



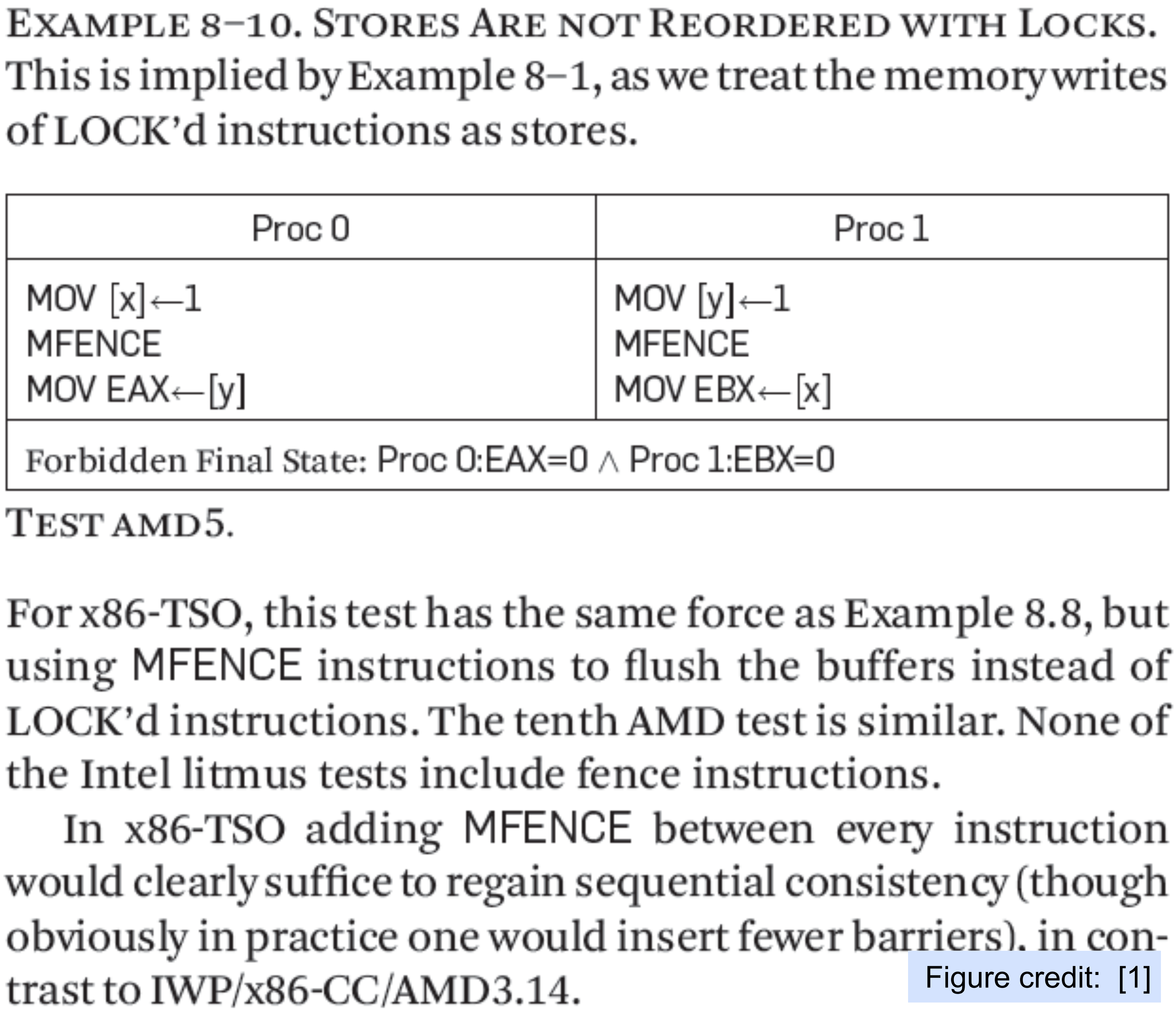
EXAMPLE 8–9. LOADS ARE NOT REORDERED WITH LOCKS.

Proc 0	Proc 1
XCHG [x]←EAX MOV EBX←[y]	XCHG [y]←ECX MOV EDX←[x]
Initial state: Proc 0:EAX=1 ∧ Proc 1:ECX=1 (elsewhere 0)	
Forbidden Final State: Proc 0:EBX=0 ∧ Proc 1:EDX=0	

This test indicates that locking both writes in Example 8–3 would forbid the nonsequentially consistent result. x86-TSO forbids the final state because LOCK'd instructions flush the local store buffer. If only one write were LOCK'd (say the write to x), the Example 8–3 final state would be allowed as follows: on Proc 1, buffer the write to y and execute the read x, then on Proc 0 write to x in shared memory then read from y.

Figure credit: [1]

x86-TSO Behaviors - IX



EXAMPLE 8–10. STORES ARE NOT REORDERED WITH LOCKS. This is implied by Example 8–1, as we treat the memory writes of LOCK'd instructions as stores.

Proc 0	Proc 1
MOV [x]←1 MFENCE MOV EAX←[y]	MOV [y]←1 MFENCE MOV EBX←[x]
Forbidden Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

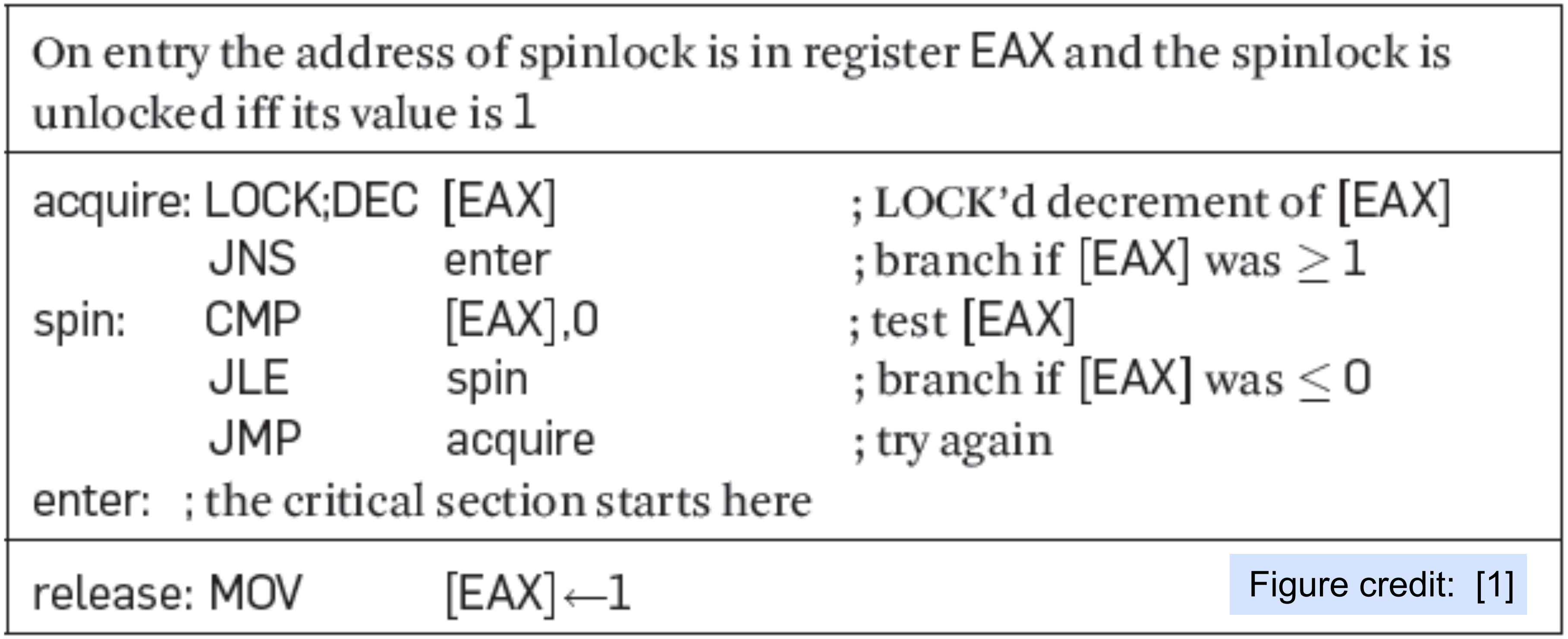
TEST AMD5.

For x86-TSO, this test has the same force as Example 8.8, but using MFENCE instructions to flush the buffers instead of LOCK'd instructions. The tenth AMD test is similar. None of the Intel litmus tests include fence instructions.

In x86-TSO adding MFENCE between every instruction would clearly suffice to regain sequential consistency (though obviously in practice one would insert fewer barriers), in contrast to IWP/x86-CC/AMD3.14.

Figure credit: [1]

Semantics of Linux Spin Locks


On entry the address of spinlock is in register EAX and the spinlock is unlocked iff its value is 1

```
acquire: LOCK;DEC  [EAX]        ; LOCK'd decrement of [EAX]
         JNS       enter        ; branch if [EAX] was ≥ 1
spin:    CMP       [EAX],0      ; test [EAX]
         JLE       spin         ; branch if [EAX] was ≤ 0
         JMP       acquire      ; try again
enter:   ; the critical section starts here

release: MOV       [EAX]←1
```

Figure credit: [1]

- **Question about Linux spin locks: is it OK to have the MOV in release as an unlocked operation?**
  - —lets releasing thread continue without flushing write buffer
- **Answer: YES!**
  - —by TSO, the stores within the critical section will all drain from the store buffer before the write that releases the spinlock

Take Away Points

- Looser HW memory models improve performance
  - —operation latency can be overlapped with other operations
- HW memory models today are loose in many ways
  - —operations within a thread may appear out of order
  - —operations by different threads may only be partially ordered
- The x86-TSO model provides an understandable model for programming x86 systems
  - —better than prior specifications, which were wrong in different ways

References

1. Sewall et al. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. CACM 53(7):89-97. http://doi.acm.org/10.1145/1785414.1785443
2. Paul E. McKenney. Memory Barriers: a Hardware View for Software Hackers, July 23, 2010. http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf
3. Scott Owens, Susmit Sarkar, Peter Sewell. A Better x86 Memory Model: x86-TSO. Theorem Proving in Higher Order Logics Lecture Notes in Computer Science, volume 5674. Springer, 2009. http://dx.doi.org/10.1007/978-3-642-03359-9_27
4. M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation, and programming. Distributed Computing, 9(1):37–49, 1995.